Using graph theory for automated electric circuit solving

L Toscano, S Stella, and E Milotti

Dipartimento di Fisica, Università di Trieste, Via Valerio 2, I-34127 Trieste, Italy

E-mail: licia.toscano91@gmail.com

Abstract. Graph theory plays many important roles in modern physics, and in many different contexts, which span such diverse topics as the description of scale-free networks, up to the structure of the universe as a complex directed graph in causal set theory. Graph theory is also ideally suited to describe many concepts in computer science. Therefore it is increasingly important for physics students to master the basic concepts of graph theory. Here we describe a student project where we develop a computational approach to electric circuit solving which is based on graph theoretic concepts. This highly multidisciplinary approach combines abstract mathematics, linear algebra, the physics of circuits and computer programming, to reach the ambitious goal of implementing automated circuit solving.

Submitted to: *Eur. J. Phys.*

1. Introduction

Networks and graphs are well-established elements of many "new physics" approaches, and they turn out in many fast-developing fields like the theory of computer networks, Markov chains, biochemical reaction networks, the statistical mechanics of disordered media, neuronal networks, food webs, traffic networks, and social networks [1, 2, 3, 4, 5, 6, 7, 8]. Unfortunately, the important concepts of graph theory are still missing from many undergraduate curricula. However, electrical networks are an important part of the standard physics curricula, and they provide an easy access route to the fascinating world of graphs and networks [8]. Indeed, graph-theoretical concepts are at the very basis of many programs for the resolution of electrical and electronic circuits – like, e.g., SPICE – and here we describe a student project to construct from scratch one such program. In this paper we give a bare-bones description of the project, and suggest a few possible developments, with the hope that we may be able to kick off a number of interesting projects at the interface of physics and graph theory.

2. Graphs

The approach that we suggest in this paper requires only a basic knowledge of graph theory. There are many excellent introductions to the theory of graphs [9, 10], and the instructor should follow her/his preferences in the exposition of graph theory. In this section we review the basic definitions and concepts that are necessary and useful for a proper development of the student project.

A *graph* is a collection of points (called *vertices* or *nodes*) and lines joining some of these points (called *arcs* or *links*). The links can be either *directed* or not, and the corresponding graphs are *directed* or *non-directed*. Figure 1 shows examples of direct and non-directed graphs.

Graphs are often a practical way to classify relationships among objects, or to describe topological properties of discrete sets, and we are interested in their overall shape. To this end it is important to introduce some standard definitions:

path : a sequence of links in a directed graph, where the final vertex of one link is the initial vertex of the next one;

chain : the counterpart of the path in a non-directed graph; for non-directed graphs paths and chains are equivalent;

circuit : a path such that the final vertex of the last link coincides with the initial vertex of the first link;

cycle : a non-directed circuit; for non-directed graphs the concepts of circuit and cycle are equivalent;

complete graph : this is a graph where each vertex is joined by an arc to every other vertex;

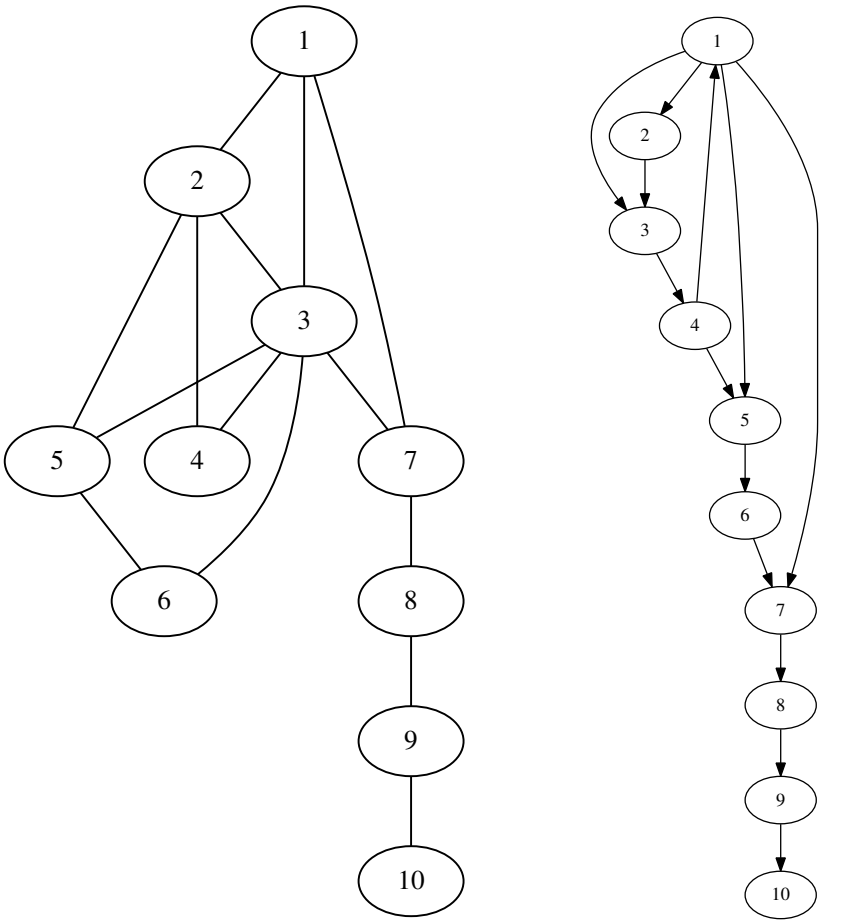

Figure 1: Examples of graphs. The graph on the left is a non-directed graph, while the graph on the right is a directed graph. Both graphs contain cycles and circuits, e.g., in the non-directed graph the chain {1, 2, 3, 1} is a cycle. Both figures have been generated with the freeware program Graphviz[12].

partial graph : a graph that contains all the vertices of the original graph but only part of its links;

subgraph : a graph that contains only a subset of the vertices of the original graph and all of the links that join them;

planar graph : a graph that can be drawn on a plane in such a way that no two arcs intersect.

In all applications – and electrical networks are no exception – it is very important to consider the connectivity properties of graphs. This can be made precise by the list of connections among vertices, which can be described by the *adjacency matrix*. For a graph with N vertices, the adjacency matrix is an $N \times N$ square matrix where each row and each column correspond to a vertex. The (i, j)-th matrix element is 1 if there is at least one arc joining the i-th and the j-th vertices and 0 otherwise.

connected graph : a graph is connected if there is at least one chain from any vertex

to every other vertex;

strongly connected graph : a directed graph is strongly connected if there is at least one path from every vertex to every other vertex;

weakly connected graph : a directed graph is weakly connected if it is not strongly connected, but there is at least one chain from every vertex to every other vertex;

tree : a tree is a non-directed, connected graph, with no circuits;

forest : a forest is any non-directed graph without circuits (in the case of unconnected graphs, a forest is a collection of trees);

spanning tree : a spanning tree of a connected graph G is a tree which is a partial graph of G.

This very basic description of graphs already leads to many interesting problems, for instance in several applications it is important to determine whether there are subgraphs that are connected to the other vertices but are not biconnected – a subgraph is biconnected when there are at least two different chains joining it to the rest of the graph. When links are endowed with numerical weights, many new important problems arise, like for instance the search for optimal paths (paths of shortest total weight).

Here we are concerned with the search and enumeration of all cycles in a graph that represents an electric circuit.

2.1. Electric circuits and graphs

The relation between Kirchhoff's method for constructing the equations that describe an electric circuit and graphs was first clarified by Veblen in 1916, in the wake of Poincaré's formal description of graphs by means of the incidence matrix [8] – which is a close relative of the adjacency matrix. Modern electronic simulation programs determine the circuit equations by means of nodal analysis [11]; here we turn to a different kind of analysis, which goes back to the tools introduced by Poincaré and Veblen, and which shows in a more direct way the relation between graphs and electric circuits. In particular we select a fundamental set of cycles – i.e., current loops – as suggested by Gotlieb and Corneil [13], without resorting to more efficient, but more complex implementations, like that of Tiernan [14]. In the following sections we describe our implementation of the algorithm, step by step.

3. Setting up Kirchhoff's equations

The analysis of electric circuits usually requires Kirchhoff's laws for both currents and voltages, however, when we define the loop currents for all the independent loops, and use the superposition principle, we only need Kirchhoff's law for voltages, because the nodal equations for currents are automatically satisfied by the superposition of loop currents.

To carry out this program we must execute the following steps:

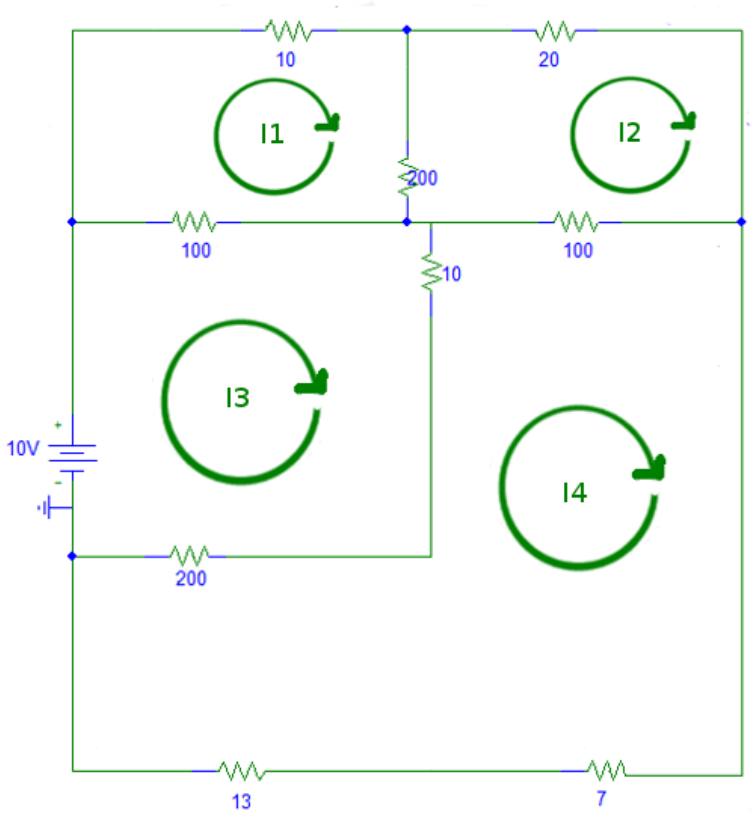

Figure 2: Example circuit. Here we show a set of independent loops with the associated loop currents. Although the units of resistance are not shown, they are conventionally assumed to be ohms while voltage values are in volt.

(i) we describe the circuit in a computer-readable format;

(ii) we find a set of independent loops;

(iii) we associate a loop current to each loop; the current orientation is arbitrary, as long as the loop does not include semiconducting junctions;

(iv) we use Kirchhoff's law for voltages to set up a system of n equations, one for each of the n independent loops;

(v) we solve the system of equations and thus find the values of the loop currents;

(vi) we use the superposition principle to find the total current in each circuit branch.

Figure 2 displays an example of a set of independent loops and the related loop currents. For each independent loop we have a corresponding equation that involves the loop current.

In the case of a purely resistive circuit like that shown in figure 2, the total voltage drop $E^{(b)}$ across each branch b is the sum of the voltage drops due to the loop currents $I_{\ell'}$ that traverse that branch, i.e.,

$$E^{(b)} = \sum_{\ell' \in L_b} R^{(b)} I_{\ell'} \tag{1}$$

where L_b is the set of loops that include branch b, ℓ' is one of the loops in L_b , and $R^{(b)}$ is the resistance associated to branch b. Then, summing over all branches of a given independent loop ℓ, we find

$$\sum_{b\in\ell} E^{(b)} = \sum_{b\in\ell} \sum_{\ell'\in L_b} R^{(b)} I_{\ell'} \tag{2}$$

For instance, when we take the circuit shown in in figure 2 we obtain the equations

$$\begin{cases} (10+200+100)I_1 - 200I_2 - 100I_3 + 0I_4 &= 0 \\ -200I_1 + (20+100+200)I_2 + 0I_3 - 100I_4 &= 0 \\ -100I_1 + 0I_2 + (100+10+200)I_3 - (200+10)I_4 &= 10 \\ 0I_1 - 100I_2 - (10+200)I_3 + (100+7+13+200+10)I_{m_4} &= 0 \end{cases}$$

where resistor values are in ohm, current values in ampere, and voltage values in volt, or equivalently, in matrix notation,

$$\begin{pmatrix} 310 & -200 & -100 & 0 \\ -200 & 320 & 0 & -100 \\ -100 & 0 & 310 & -210 \\ 0 & -100 & -210 & 330 \end{pmatrix} \begin{pmatrix} I_1 \\ I_2 \\ I_3 \\ I_3 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 10 \\ 0 \end{pmatrix}$$

4. The algorithm

This section discusses the algorithmic implementation of the steps listed in section 3. In our project we decided to use the C programming language because of its widespread diffusion, because of the availability of many mathematical libraries, and also because of the easy portability into C++ code for further developments. Obviously this choice is not binding, and in the following sections we list relevant pseudocode rather than C code.

4.1. Circuit's description

All the required information about the circuit is in a data file which is the input file of the program. We describe an electric circuit with n nodes with three $n \times n$ matrices: the adjacency matrix $A = \{a_{ij}\}$, the resistors' matrix $R = \{r_{ij}\}$ and the voltage sources' matrix $V = \{v_{ij}\}$. As usual, in the adjacency matrix the element $a_{ij} = 1$ when nodes i and j are connected, while $a_{ij} = 0$ when the nodes are not connected. Likewise in the resistors' matrix the element r_{ij} is equal to the resistance of the branch that links nodes i and j, and in the voltages' matrix the element v_{ij} is equal to the voltage of a voltage source in the branch that links nodes i and j. The sign convention for the voltage is that the matrix element v_{ij} is positive if node i has a lower potential than node j. The total number of nodes, as well as the three matrices are stored on file and are read by the program during the initialisation step. Figure 3 shows one enumeration of the n nodes in the circuit of figure 2: notice that when a branch contains more than one element we split it by adding one or more nodes (such as node 10 in figure 3).

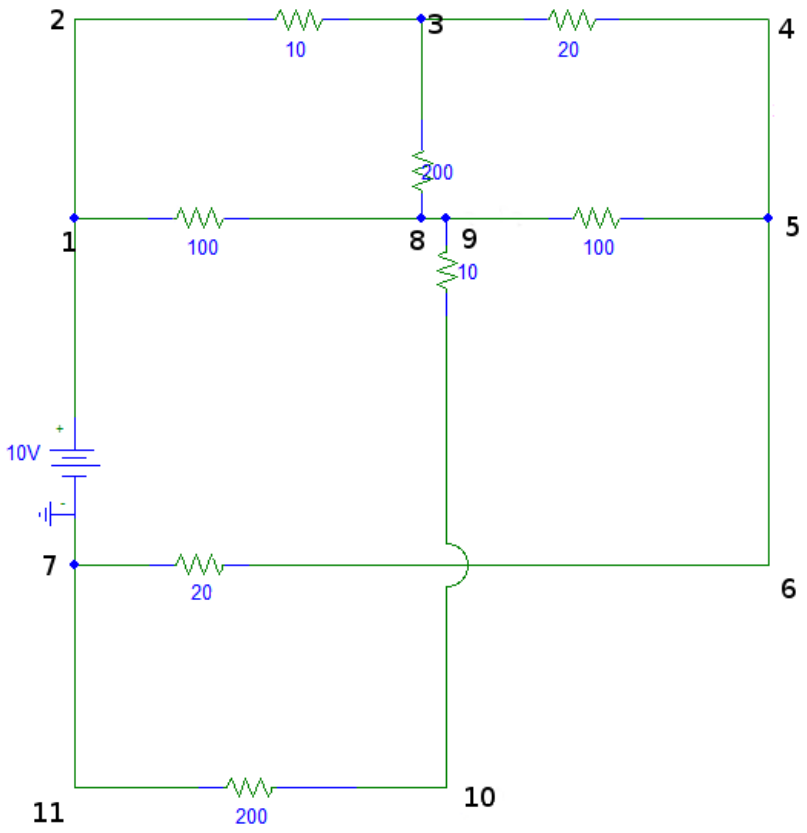

Figure 3: One possible enumeration of the nodes in the circuit of figure 2.

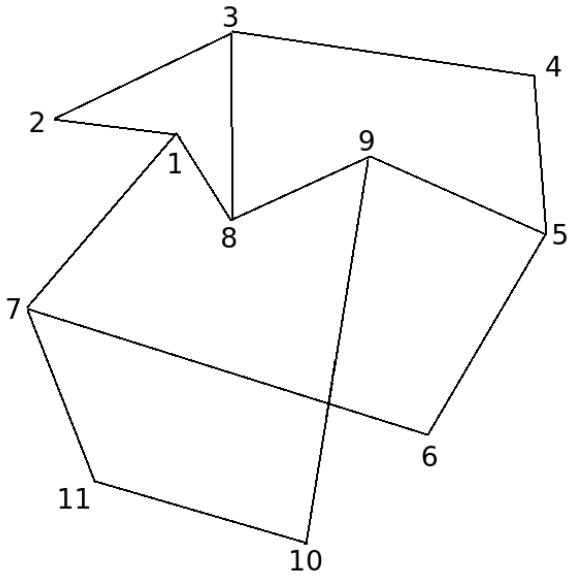

Figure 4: Graph corresponding to the circuit in figure 2.

$$\begin{pmatrix}
0 & \mathbf{1} & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\
\mathbf{1} & 0 & \mathbf{1} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & \mathbf{1} & 0 & \mathbf{1} & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & \mathbf{1} & 0 & \mathbf{1} & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & \mathbf{1} & 0 & \mathbf{1} & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & \mathbf{1} & 0 & \mathbf{1} & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 & \mathbf{1} & 0 & 0 & 0 & 0 & \mathbf{1} \\
1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & \mathbf{1} & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & \mathbf{1} & 0 & \mathbf{1} & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{1} & 0 & \mathbf{1} \\
0 & 0 & 0 & 0 & 0 & 0 & \mathbf{1} & 0 & 0 & \mathbf{1} & 0
\end{pmatrix}$$

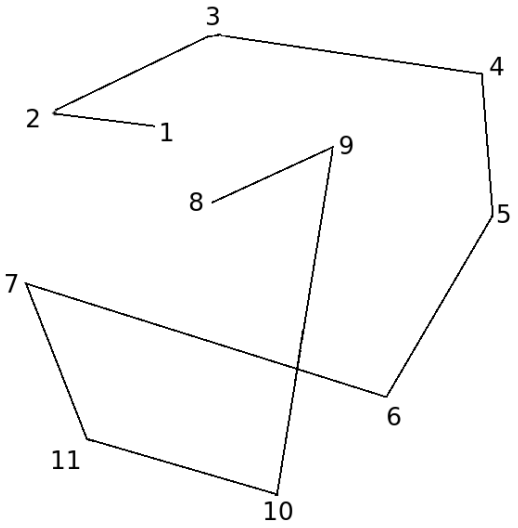

Figure 5: The left panel shows the adjacency matrix of the graph in figure 4; when we keep only the boldface 1's, we obtain the spanning tree shown in the right panel.

4.2. Finding a set of independent loops

In order to set up a system of independent equations for the currents, we need a *fundamental set of cycles*‡, i.e., a set of independent current loops. Here follow the hint in the paper of Gotlieb and Corneil [13], i.e., the algorithm works by constructing a spanning tree first, and then it yields a fundamental set of cycles by adding the discarded branches, one at a time, to the spanning tree.

4.2.1. Construction of the spanning tree We implement a vertex-centric algorithm to find a spanning tree, as shown in the pseudocode is shown in Box 1 (see, e.g., [15]), and figure 5 shows the A and B adjacency matrices and the resulting spanning tree.

4.2.2. Construction of the independent loops At this stage we have a spanning tree and a set of leftover links. Starting from the spanning tree, it is easy to obtain the independent loops: it suffices to add one of the discarded links to the spanning tree, one at a time. Next it is necessary to prune the tree to get rid of the remaining branches. Figure 6 shows a simple example, and Box 2 lists the pseudocode that carries out these steps (we omit explicit coding of the pruning step: it is a straightforward but rather tedious part, and we leave it as an exercise to the reader).

‡ The cycles of a fundamental set are such that: **a**) their union contains all the cycles of the original graph, and **b**) they are independent, in the sense that any given cycle in the set is not contained in any union of the remaining cycles.

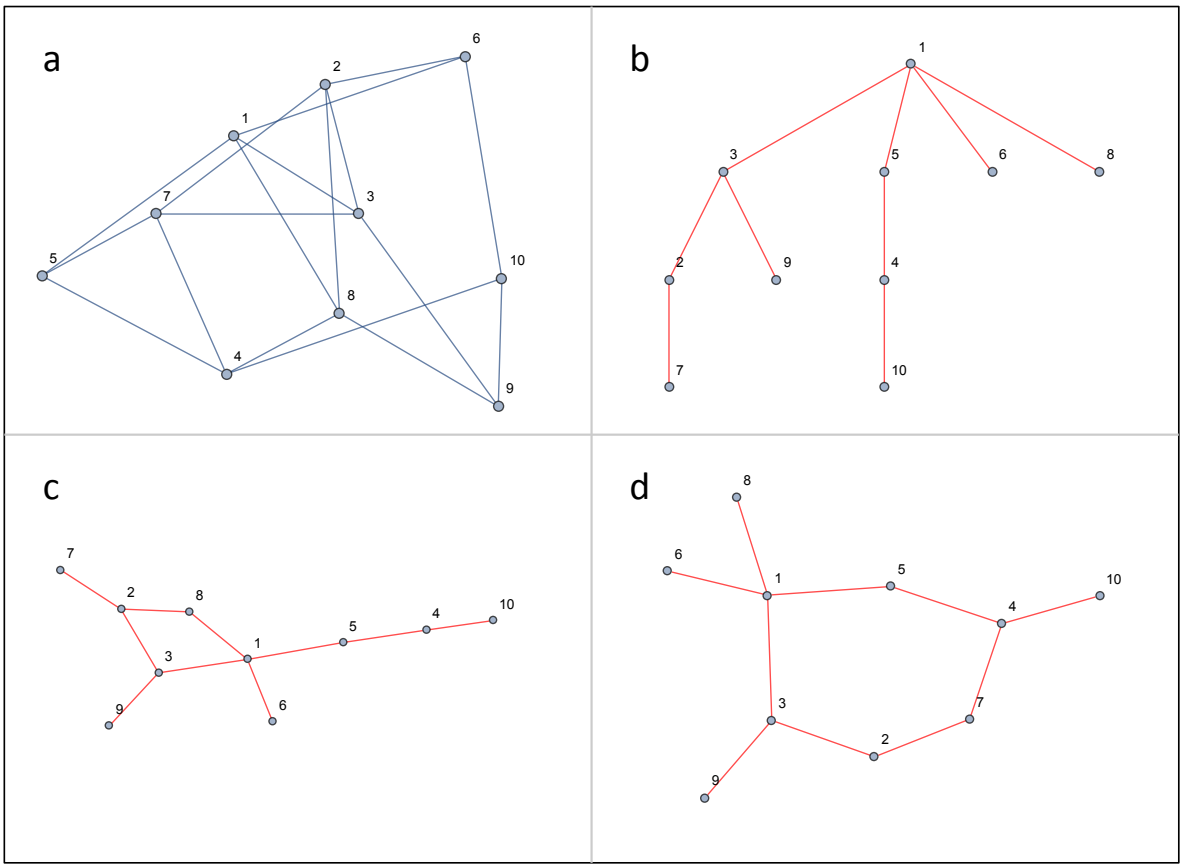

Figure 6: Finding independent loops. Panel **a** shows an example graph with 10 nodes. Panel **b** shows one spanning tree for the graph in panel **a**. Panels **c** and **d** show two graphs obtained from the spanning tree by adding the link between nodes 2 and 8 that was discarded in the construction of the spanning tree (panel **c**), and the discarded link between nodes 4 and 7 (panel **d**). All that is left to do is to prune the graphs in panels **c** and **d** to keep the loops and get rid of the branches.

Box 1: pseudocode to construct a spanning tree with the vertex-centric algorithm.

```
C is a vector of size N and is the list of vertices already included
        in the spanning tree
A, B are matrices of size N × N;
        A is the original adjacency matrix,
        B is the adjacency matrix of the spanning tree

/*here we initialise C and B */
C_0 ⟵ 1

for i, from 1 to N − 1
   C_i ⟵ 0

for i, j from 0 to N − 1
   B_ij ⟵ 0

```

```
/* here we construct the spanning tree */
for i,j from 0 to N-1
  if A_ij = 1 and C_i = 1 and C_j = 0
    B_ij ⟵ 1
    B_ji ⟵ 1
    C_j ⟵ 1
  if A_ij = 1 and C_i = 0 and C_j = 1
    B_ij ⟵ 1
    B_ji ⟵ 1
    C_i ⟵ 1
```

Box 2: pseudocode to find a set of fundamental cycles.

```
m is the number of fundamental cycles

A, B, C, D^(k) are matrices of size N×N (and 0≤k≤m-1);
        A is the original adjacency matrix,
        B is the adjacency matrix of the spanning tree
        D^(k) is the adjacency matrix of the spanning tree, augmented
          with the addition of the k-th discarded link

/* first we find the set of discarded links */
/* the C matrix has 1's in the positions of the discarded links */
/* we also copy the B matrix into the D matrix */
for i from 0 to N-1
  for j from 0 to N-1
    C_ij ⟵ A_ij - B_ij
    D_ij^(k) ⟵ B_ij

/* here we find all the cycles of the fundamental set */
k ⟵ 0
for i from 0 to N-2
  for j from i+1 to N-1
    if C_ij = 1 /* here we find a discarded link between nodes i and j */
      k ⟵ k+1
      D_ij^(k) ⟵ 1
      D_ji^(k) ⟵ 1
      prune the D^(k) matrix

m ⟵ k
```

4.3. Loop orientation

Having found a fundamental set of cycles, i.e., a set of independent current loops, we must set the loop orientation, i.e., the arbitrary direction of current flow. When we set the loop orientation we also change the graph from undirected to directed, and we must modify the adjacency matrix A accordingly, i.e., while in the undirected case both A_{ij} and A_{ji} equal 1 when a link exists between nodes i and j, only one of the two matrix elements is actually nonzero in the undirected case, and the A matrix is no

longer symmetric. Here we proceed as follows:

(i) we loop over all cycles;

(ii) we select one link in the loop and delete its symmetrical; this choice is arbitrary, and it sets the loop orientation;

(iii) following the direction of the first link, we set the direction of the other links in the loop;

(iv) return to step (i), or stop if all the loops have been oriented.

4.4. Circuit equations

After having found and having oriented the independent loops, we can set up the system of equations (1). Here we are mainly concerned with simple linear circuits – with either resistances or impedances – and therefore we need a C library for the solution of linear systems. We have chosen the algorithm based on the LU decomposition method (routines `ludcmp` and `lubksb`) in the Numerical Recipes C library [16], because it is quite well documented and simple to include in our code, although many other alternatives are actually available, and can easily replace it in other implementations of this code.

4.5. Branch currents and voltages

The solution of the linear equations yields all the independent loop currents, and from them we find the branch currents by summing over all the independent loop currents that traverse a given branch. By looping over all branches we can construct a new $n \times n$ matrix $I = \{i_{jk}\}$ where the matrix element i_{jk} stores the current in the branch that links the j-th node with the k-th node.

Likewise, we calculate a voltage matrix $V = \{v_{jk}\}$, where the matrix element $v_{jk} = r_{jk} i_{jk}$ stores the voltage drop in the branch that links the j-th node with the k-th node.

Voltages are set by the power supplies with respect to a ground terminal: we can find the voltage of any node with respect to ground simply by following any path along the circuit that joins the node with the ground node and summing all the voltage drops.

In a final step of the algorithm, the results are stored on file and saved for further analysis.

5. Comparison with SPICE

As stated at the beginning, this student project is meant to be an instructional path through the mathematics of graphs and basic computer coding, and it is not aimed at an efficient code rivalling established circuit solvers like SPICE (Simulation Program with integrated Circuit Emphasis) [17]. However, it is interesting to compare the results with

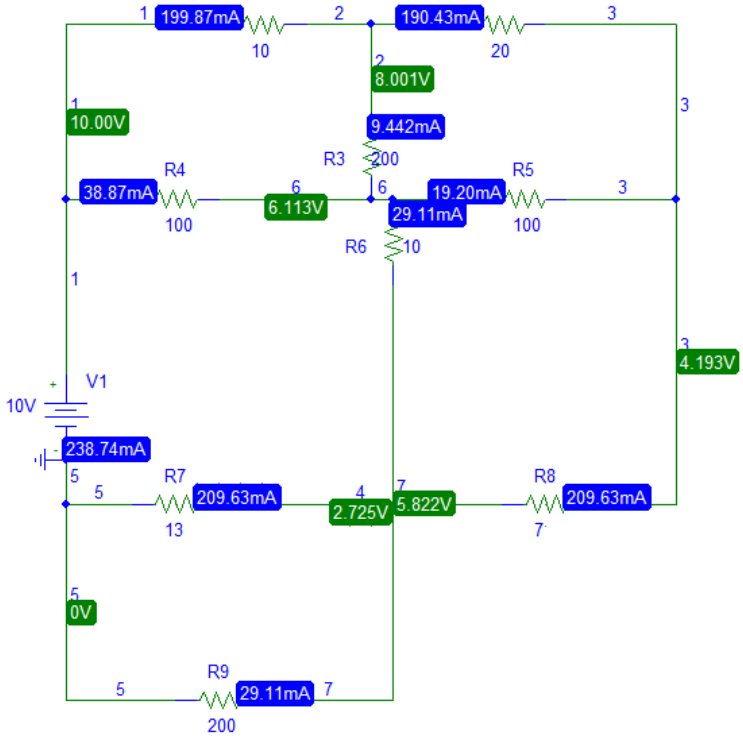

Figure 7: PSpice's output for the example circuit of figure 2.

those of obtained with SPICE or one of its many commercial versions, like PSPICE, which has a beautiful graphical interface and can be downloaded for free for student use [18]. Figure 7 shows the PSPICE output for the example circuit of figure 2: the figure lists currents and voltages, and these numbers coincide, up to the numerical precision of the code, with the results obtained by our program.

6. Conclusions

We believe that the student project described in this paper is a strongly motivating introduction to intermediate-level programming, because of its relative complexity and the need to construct efficient computational structures. It is also a sort of "work in progress" and represents a first step in the direction of more complex circuit solving programs. Therefore it can accommodate the efforts of many more students, who could improve the program in many ways, for instance:

- by performing an accurate comparison of the present graph-based approach with the nodal methods used in established programs like SPICE [11, 17];
- by optimising the circuit representation in the description file;
- since all the matrices are sparse, the program efficiency could be boosted with the use of sparse matrix code;
- the code could be cleaned up and improved with the introduction of linked lists to represent loops and other graph structures;

- the program could be extended to handle time-dependent voltages and currents; instead of the system (1) we should manipulate the corresponding differential equations and use a differential equation solver;
- the program could also be extended to include non-linear elements like semiconductor diodes, transistors, etc., and this would require in turn the introduction of a nonlinear equation solver.
- the program could be modified using objects and the powerful concepts of object-oriented programming: this would open up new possibilities, also because of the availability of efficient C++ libraries like Boost [19].
- ambitious students could work on a graphical interface both for input – by placing components in a special window – and for output – by automatically plotting output functions.
- the same ideas could be expanded to produce a graphical tool to manipulate and display graphs, as an aid in understanding the modern research on networks [20].

The potential of the scheme described in this paper is witnessed by the fact that the computer code written during the student project at the Physics Dept. of the University of Trieste is flexible enough to help another student in its analysis of fluid flow – using the analogy between electric and hydraulic circuits (see, e.g., [21]). On the whole, we believe that this is a highly motivating open project, which can stimulate interest not just in the physics of electric circuits, the mathematics of graphs, and in computational physics, but in its many applications as well.

References

[1] Hayes B 2000 Graph Theory in Practice: Part I *Am. Sci* **88** 9-13
[2] Hayes B 2000 Graph Theory in Practice: Part II *Am. Sci* **88** 104-109
[3] Albert R and Barabási A-L 2002 Statistical mechanics of complex networks *Rev. Mod. Phys* **74** 47-97
[4] Dorogovtsev S N and Goltsev A V 2008 Critical phenomena in complex networks *Rev. Mod. Phys* **80** 1275-1335
[5] Barabási A-L and Oltvai Z N 2004 Network biology: understanding the cell's functional organisation *Nature Rev. Genetics* **5** 101-113
[6] Estrada E 2013 *Graph and Network Theory in Physics* arXiv:1302.4378
[7] Erciyes K 2013 *Distributed Graph Algorithms for Computer Networks* (New York, NY: Springer)
[8] Cederbaum I 1984 Some Applications of Graph Theory to Network Analysis and Synthesis *IEEE Trans. on Circuits and Systems* **31** 64-68
[9] Ore O 1990 *Graphs and their uses* (Washington, DC: Mathematical Association of America)
[10] Gould R 1988 *Graph Theory* (Menlo Park, CA: The Benjamin/Cummings Publishing Company)
[11] Vlach J and Singhal K 1983 *Computer Methods for Circuit Analysis and Design* (New York, NY: Van Nostrand Reinhold)
[12] Gansner E R and North S C 2000 An open graph visualization system and its applications to software engineering *Software - Practice and Experience* **30** 1203-1233
[13] Gotlieb C C and Corneil D G 1967 Algorithm for Finding a Fundamental Set of Cycles for an Undirected Linear Graph *Comm. of the ACM* **10** 780-783

[14] Tiernan J C 1970 An Efficient Search Algorithm to Find the Elementary Circuits of a Graph *Comm. of the ACM* **13** 722-726
[15] Nicos C 1975 *Graph Theory, an Algorithmic Approach* (London: Academic Press)
[16] Press W H, Teukolsky S A, Vetterling W T, and Flannery B P 1997 *Numerical recipes in C: The Art of Scientific Computing, 2nd Edition* (Cambridge: Cambridge University Press)
[17] Nagel L W and Pederson D O 1973 SPICE (Simulation Program with Integrated Circuit Emphasis) ERL-M382 (Electronics Research Laboratory – UCLA, Berkeley)
[18] `http://www.electronics-lab.com/downloads/schematic/013`
[19] `http://www.boost.org`
[20] Barabási A-L and Bonabeau E 2003 Scale-Free Networks *Scientific American* May 2003, 50-59
[21] Westerhof N, Stergiopulos N, and Noble M I M 2010 *Snapshots of Hemodynamics, An Aid for Clinical Research and Graduate Education, 2nd edition* (New York, NY: Springer)